Appendix T
Software Testing

T–1. Overview of software testing

This appendix briefly describes a variety of software testing methods that can be used to detect errors, develop sets of test data, and monitor computer system resources. These methods serve only as a representative sample, rather than an all-inclusive list. See appendix Q, which also addresses DT of software.

T–2. Software test limitations

a. The objective of functional or "path" testing is to use known input data to check whether the output functions meet established specifications. Each piece of input data will generate a software function along a specific path of digital logic. However, the technology of real-time, embedded software places strict limitations on the number of software elements that can actually be exercised in a functional test.

b. The most significant technical difficulty in path testing is to create test cases that will provide adequate coverage of the software paths. Success of path testing is determined by the degree of coverage that is achieved. However, many test data sets will produce either redundant paths or paths that are infeasible because they violate design constructs of the software. Therefore, selecting an adequate set of test cases requires specialized support tools, even for non-complex programs.

c. The difficulty in achieving adequate test coverage also prohibits use of a single, OT to certify software acceptance. Most embedded software programs are large, containing many modules and decision statements that may produce different outputs with the same inputs but with slight variations in execution times. Identifying the actual source of an error in an OT is extremely difficult.

d. A longer test is also no guarantee of adequate coverage, due to unequal distribution of modular run times. Most disciplined software development processes use a modular, hierarchical approach to design and test software. Top-level modules provide functional control over the lower-level modules that they call for. Therefore, the top-level software modules are exercised much more frequently during integration testing than the lower-level modules that await calls. Various estimates of run-time distributions have documented that only 4 to 10 percent of software architecture will operate for 50 to 90 percent of the total run time. Therefore, increasing the length of a test may only fractionally expand coverage of software functions.

T–3. Software incremental testing

An incremental test strategy allows a variety of test events that are diverse enough to provide confidence in the effectiveness of the test process. In addition, an incremental strategy provides a means to identify and correct failures earlier and more effectively. Specific test events and levels are tailored to the needs of each system acquisition.

T–4. Software testing techniques

Testing takes place at various points of the software development process that are generally common to all software projects. They are—

a. Unit Testing in which each unit, or basic component, of the software is informally tested to verify that the detailed design for the unit has been correctly implemented. Unit testing validates each program unit in isolation. The tests are usually performed by the programmer who designed and coded the unit.

b. Software Integration Testing in which progressively larger groups of tested software units are integrated and tested until the software works as a whole.

c. System Testing in which the software is integrated with the overall product and tested to verify that the system meets its specified requirements.

d. Acceptance Testing generally involves a subset of system tests that formally demonstrates key functionality for final approval and contract compliance. Acceptance testing is witnessed by the customers; it may be performed at the developer's site or at the user's site. Each of these four test stages make use of static and dynamic analysis techniques that are described in paragraphs T–5 and T–6.

T–5. Static software analysis

Static analysis examines or analyzes a software product without executing the code on computer or system hardware. Instead, static analysis is a manual task or an automated process using static, source code analysis tools. Static analysis tools can demonstrate the absence of certain types of defects, such as variable typing errors, but they cannot alone detect faults that depend on the underlying operating environment. Consequently, effective software testing requires a combination of static and dynamic analysis approaches. Static analysis techniques include the following:

a. Reviews, walk-throughs, and code inspections examine design and technical documentation to detect errors. The procedure typically involves a small working group of programmers and technical personnel who assess requirements documents, design specifications, and program listings. This static analysis procedure is an essential task in software development. It is commonly referred to as peer review, which is one of the key process areas that a developer must perform to achieve level 3 through 5 in the Capability Maturity Model (CMM). Peer review may include an individual

or group analysis of design logic representations, a line-by-line code reading, analysis of documentation on test inputs, or tracing requirements from document to document.

b. Code audits examine the source code to determine whether prescribed programming standards and practices have been followed.

c. Interface checking examines the flow of information and control within a system to identify areas where mistakes can occur, such as calling the wrong procedure or passing the incorrect data.

d. Physical units checking specify and checks measurement units in computations.

e. Data flow analysis detects whether or not sequential events occur in software execution.

f. Structure analysis detects violations of control flow standards, such as improper calls to routines, infinite loops, and incidents of recursion in design products or source code.

g. Cross reference checking produces lists of data names and statement labels to show all places they are used in a program.

h. Input space partitioning uses path and domain analysis, or partitions to build sets of test data that cause a selected path in software to be executed.

i. Complexity analysis examines algorithm design or coded programs to examine the density of decision options, number of operations required, amount of capacity used, or understandability of the code.

T–6. Dynamic software analysis

a. Dynamic analysis executes the software to determine if it functions as expected. Dynamic analysis may involve running the software in a special test environment with stubs, drivers, simulators, test data, or it may use an actual operating environment with real data and operational conditions. Current tools attempt to detect faults rather than demonstrate their absence. Additionally, most of these tools can only detect faults that extend to software outputs, unless the software has been specially instrumented to monitor internal data elements (intrusive monitoring) or special hardware monitors have been attached to the system (non-intrusive monitoring). Most importantly, the effectiveness of any dynamic analysis technique is directly related to the quality of the test data.

b. Proper selection of input data must be based on an accurate description of the design of the computer program and host system. In most large-scale, software development programs, accurate design information may be best derived through the Verification and Validation (V&V) effort. The objective of software V&V prior to functional testing is to ensure that the software design conforms to established specifications and that the design and code are free of errors. Software must conform to requirements specifications at each level of the system to allow proper assessment of system outputs. Software functions can be verified as correct only if the observed system output is in compliance with the intent of the test case input. Specifications must be written with a level of detail to allow verification of proper input/output relationships at every level in the system. The V&V process will also provide the technical insight to program design and behavior that is required to structure an effective stress test program.

c. Dynamic analysis techniques include the following:

(1) Functional (black box) testing is the most commonly used dynamic analysis approach. This approach executes the program with specific, controlled input to verify that the program performs the correct functions. For functional strategies, test data are derived from program requirements without regard to program structure. The amount of software that can be exercised in a functional test is limited by the test environment and the time available for testing. Therefore, use of this method alone does not guarantee a thorough test of the software source code or an absence of errors.

(2) Structural (white box) testing requires knowledge of the source code, including program structure, variables, or both. In structural strategies, test data are derived from the software program's structure. This approach executes the software program with specific, controlled inputs to provide a degree of coverage for the control paths, data paths, and conditions within the software program.

(3) Real-time testing, or stress testing is performed to ensure that software will support the system under the stress levels that are expected in the actual operating environments. These tests are often structured to go beyond the expected conditions to determine points where the software operation will cause system failure. Test configurations that may be used in structuring a software stress test are as follows:

(a) Excessive system functional loads that are required to support tactical operations.

(b) Extreme software inputs or conditions that cause extreme outputs.

(c) "Illegal" data inputs or conditions that replicate operator-induced input errors or equipment errors under field stress.

(d) High loading of computer capacities, including storage and processing utilization.

(4) Assertion testing uses an assertion preprocessing tool to specify and assess the intent of input, output, intermediate steps of functions, and constraints.

(5) Model-based testing is used to systematically select a set of test case inputs and outputs that have a high probability of detecting existing errors.

(6) Performance measurement techniques monitor software execution to locate code or throughput inefficiencies, either by random sampling or by means of software probes.

(7) Path and structural analysis monitors the number of times a specific portion of code is executed, the amounts of time involved, and other design parameters to detect errors in computation, logic, data handling, or output.

(8) Interactive debugging techniques control program execution and analyze any part of a program while it executes.

(9) Random testing can reveal unexpected program behavior by executing the program with random data and comparing the actual output to the expected output.

(10) Mutation analysis studies the behavior of many versions of the same program that have been mutated with a number of errors to check that each mutant produces different output data when given the same input data.

(11) Error seeding uses the percentage of detected errors to extrapolate the estimated number of remaining errors in a large software system.

T–7. Security certification

The software test program must accommodate the requirements of AR 380–19 regarding information security. Examining the control of the procedures used during design and test to develop software is an integral part of the software certification and system accreditation process.

a. Software must be completely tested before becoming operational.

b. Both valid and invalid data must be used for testing.

c. Testing is not complete until all security mechanisms have been examined and expected results attained.

d. Upon completion of maintenance or modification of software, independent testing and verification of the changes is required before returning the software to operation.

T–8. Computer software configuration item qualification testing

a. During this activity, the developer prepares and demonstrates all the test cases necessary to ensure compliance with the CSCI software and interface requirements.

b. If a multiple build software acquisition strategy is in effect, this activity for a CSCI is not complete until that CSCI's final build, or possibly later builds involving items with which the CSCI is required to interface.

c. Historical equivalent activities are—

— CSCI formal qualification test (FQT).
— Materiel system computer resources (MSCR).
— Software development test cycle/system testing (partial).
— AIS.

d. The objective of CSCI qualification testing is to demonstrate to the acquirer the CSCI's ability to meet its requirements as specified in its software and interface requirements specifications.

e. Entry criteria can consist of—

(1) The CSCI should successfully complete unit integration and testing, including developer internal CSCI testing.

(2) Test preparation effort, including STD preparation and dry run, should occur prior to running a formal test witnessed by the acquirer.

f. Test activities include—

(1) The developer establishes test preparations, test cases, test procedures, and test data for CSCI qualification testing and records this information in the appropriate STD.

(2) Benchmark test files are used as test data, if available.

(3) Prior to an acquirer witnessed test, the developer should perform a dry run of the test in accordance with the test cases, procedures and data in the STD. The results are recorded in the appropriate SDFs and test cases or procedures are updated as needed.

(4) The developer conducts CSCI qualification testing in accordance with the test cases, procedures, and data in the STD.

(5) All discrepancies, malfunctions and errors will be documented in problem and change reports and entered into the developer's corrective action system.

(6) Results of CSCI qualification testing are recorded in a software test report (STR).

(7) Test results are analyzed, software revised and retested at all necessary levels, and the SDFs and other software products updated based on the results. The acquirer should be notified in advance when qualification retesting is to occur.

(8) The operating environment for CSCI qualification testing is usually a local test bed system. However, qualification on target or production representative system is preferred, particularly for embedded MSCR.

g. Evaluation activities are as follows:

(1) Continuous evaluation activities include—

(a) Review of the STD to ensure CSCI qualification test preparations, test cases and test procedures are adequate to verify compliance with STP and SRS/IRS requirements.

(b) Assessment of test drivers for their ability to induce data and processing loads stated in the operational mode summary/mission profile (OMS/MP). See AR 71–9 for details on the OMS/MP.

(c) Ensuring traceability from each STD test case to its CSCI and software interface requirements and, conversely, from each CSCI and applicable software interface requirement to the test case(s) that address it.

(2) Implementation and analysis of applicable metrics.

(3) If needed to resolve open issues or address areas of risk identified in the evaluation process, a formal test readiness review is appropriate.

h. The metrics marked with an *X* in table T–1 apply to CSCI qualification testing.

i. Representative products, documents and decision criteria typically addressed during CSCI qualification testing are shown in table T–2. Items marked "final" should contain comprehensive material that corresponds to the current build and level of qualification testing.

Table T–1
Metrics applicable to CSCI qualification testing

Applies	Metric
X	Cost
X	Schedule
X	Computer resource utilization
	Software engineering environment
X	Requirements traceability
X	Requirements stability
X	Design stability
	Complexity
X	Breadth of testing
X	Depth of testing
X	Fault profiles
	Reliability

Table T–2
CSCI qualification testing decision criteria

Primary responsibility	Principal products affected	Decision criteria
PM and Developer with SQA and IV&V	Test readiness review(s), if required, to resolve open issues	Ready to perform CSCI qualification test(s)
S/W Developer	STD STD STR	Draft Dry run of CSCI qual. test IAW STD Final Final
S/W Developer and Gov't. SQA or IV&V	Requirements Trace(s) Metrics Report(s)	Updated Acceptable degrees of: requirements traceability and stability; computer resource utilization; design stability; breadth and depth of testing; fault profiles